Algorithms Re-Exam
TIN093/DIT093/DIT602

Course: Algorithms

Course code: TIN 093 (CTH), DIT 093 and DIT 602 (GU)

Date, time: 25th August 2022, 14:00–18:00

Building: Johanneberg

Responsible teacher: Peter Damaschke, Tel. 5405, email `ptr@chalmers.se`

Examiner: Peter Damaschke

Exam aids: dictionary,
printouts of the Lecture Notes (possibly with own annotations),
one additional A4 paper (both sides).

Time for questions: around 15:00 and around 16:30.

Solutions: will be published after the exam.

Results: will appear in ladok.

Point limits: 28 for 3, 38 for 4, 48 for 5; PhD students: 38. Maximum: 60.

Inspection of grading (exam review): to be announced.

Instructions and Advice:

- First read through all problems, such that you know what is unclear to you and what to ask the responsible teacher.
- Write solutions in English.
- Start every new problem on a new sheet of paper.
- Write your exam number on every sheet.
- Write legible. Unreadable solutions will not get points.
- Answer precisely and to the point, without digressions.
  Unnecessary additional writing does not only cost time.
  It may also obscure the actual solutions.
- But motivate all claims and answers.
- Strictly avoid code for describing a complex algorithm.
  Instead *explain* in your words how the algorithm works.
- If you cannot manage a problem completely, still provide your approach or partial solution to earn some points.
- Facts from the course material can be assumed to be known.
  You don't have to repeat their proofs.

Remark: The number of points is not always "proportional" to the length or difficulty of a solution, but it may also be influenced by the importance of the topics and skills.

Good luck!

Problem 1 (12 points)

We are given an undirected connected graph $G = (V, E)$ and a subset $P \subset V$ of $2k$ nodes. The problem is to pair up the nodes in P, that is, form k pairs of nodes, and connect every such pair $\{u, v\}$ by some path with end nodes u and v. The only condition is that no two paths may share edges.
(Do not worry about the motivation of this problem. It appears as a subtask in some other route planning problems.)
Surprisingly, there always exists a solution, and it can be found by a greedy-like algorithm: Start with an arbitrary pairing and an arbitrary system of paths for it. Then choose two arbitrary paths that share some edge $x - y$, such as:
$s - \ldots - x - y - \ldots - t$ and $u - \ldots - x - y - \ldots - v$.
Replace them with two new paths
$s - \ldots - x - \ldots - u$ and $t - \ldots - y - \ldots - v$
which do not use the edge $x - y$ anymore. Repeat this step until no two paths share edges anymore.

1.1. Show that this algorithm always terminates and produces some valid solution in polynomial time. You need not prove a "good" specific time bound here (this would be quite laborious); it suffices to argue why the algorithm is correct and the complexity is polynomial.
Hint: It might be useful to study how the total length (number of edges) of all paths behaves. (8 points)

1.2. Finally we make the problem stricter: As before, the k paths in the solution must not share any edges. Moreover, every path in the solution must be a shortest path between its end nodes. Modify the above algorithm such that it solves this stricter problem, still in polynomial time, and motivate why your modification works correctly. (4 points)

Problem 2 (9 points)

Let $X = (x_1, x_2, \ldots, x_n)$ be a given sequence of numbers, and let J be some positive constant. A subsequence of X is any selection of numbers from X, not necessarily consecutive, but appearing in the given order. More formally, we may choose s indices $k_1 < k_2 < \ldots < k_s \leq n$, and the subsequence with these indices is then $(x_{k_1}, x_{k_2}, \ldots, x_{k_s})$. A subsequence is called monotone if $x_{k_1} \leq x_{k_2} \ldots \leq x_{k_s}$. A jump is a pair of neighbored elements in the subsequence whose difference is at least J. The problem is now to find a monotone subsequence of X with a maximum number of jumps.

Example: Let $X = (1, 3, 4, 7, 6, 12, 8, 15, 13, 18)$ and $J = 4$. Then $(1, 6, 12, 18)$ is monotone, and every pair of neighbors is a jump. Another solution with 3 jumps is $(3, 8, 13, 18)$, and there are more possible solutions. But no subsequence with 4 jumps exists.

2.1. Develop a dynamic programming algorithm for the problem. As the idea is perhaps not totally obvious, we propose already a definition of a function: For every index k, let $OPT(k)$ be the maximum number of jumps in a monotone subsequence that ends exactly with x_k. – Now describe how you compute the values $OPT(k)$ efficiently, and argue why your method is correct. You need not describe the backtracing. (6 points)

(Please follow this hint and do not attempt to find a simple greedy algorithm instead, as this is doomed to failure.)

2.2. Give and motivate a time bound for your computation. Of course, it should be a "small" polynomial bound. You can assume arithmetic operations with numbers to be elementary operations. (3 points)

Problem 3 (10 points)

Let $G = (V, E)$ be an undirected graph with n nodes and $m > n \log_2 n$ edges. Moreover, suppose that every node in V is represented by a point in the plane, with known coordinates. Let the length of any edge (u, v) in E be defined as the usual Euclidean distance of the points u and v in the plane. Now we want to find the shortest edge in E. Note that the lengths of the edges are not explicitly given, we can only compute them from the coordinates of their nodes. Since $m > n \log_2 n$, it would be nice to avoid naive computation and comparison of all edge lengths.

Take the known $O(n \log n)$-time algorithm for the Closest Points problem and modify it such that it solves the above problem within the same time bound. It suffices to explain informally what you change in the algorithm, and why this modification is correct and does not increase the time bound.

Problem 4 (15 points)

A triangle in a graph is a set of three nodes u, v, w such that all edges uv, uw, vw exist (in other words, a clique of three nodes). A graph without triangles is called triangle-free. We want to show that the Independent Set problem remains NP-complete when it is restricted to triangle-free graphs. To this end we use the following reduction. Let $G = (V, E)$ be any undirected graph with n nodes and m edges. We replace every edge (u, v) in E with a path $u - x - y - v$, where x and y are two fresh nodes. (For clarity: for every edge in E, two additional nodes are created; these are together $2m$ new nodes.) Let H denote the resulting graph.

Prove the following statements:

- H is triangle-free.
- The reduction runs in polynomial time.
- G has an independent set with at least k nodes if and only if H has an independent set with at least $k + m$ nodes.

Finally argue why these statements together imply the NP-completeness of Independent Set for triangle-free graphs.

Problem 5 (14 points)

We are given a tree T with n nodes, where every edge has some given positive length. A vehicle is able to traverse the edges of T, in either direction and arbitrarily often. Starting from some fixed node r, we want the vehicle to visit every node of T at least once and finally return to r. The goal is to minimize the total length of such a tour. We denote this problem $TOUR(T, r)$.
Let $r_1, \ldots, r_d$ denote the nodes adjacent to r, and let T_i be the subtree that would be obtained by deleting the edge rr_i, and which contains the node r_i.
The following recursive algorithm for $TOUR(T, r)$ is proposed:
For $i = 1$ to d, go from r to r_i, then perform an optimal tour that solves $TOUR(T_i, r_i)$, and go from r_i back to r.

5.1. Argue why this algorithm yields indeed a valid and optimal solution to the $TOUR(T, r)$ problem. (12 points)

Hint: It is advisable to argue in a few steps (and motivate every step):
(a) Why is the solution valid, i.e., satisfies the constraints of the problem?
(b) How often does this algorithm traverse every edge?
(c) In an arbitrary(!) valid solution, how often do we have to traverse every edge at least?
(d) Now conclude optimality. But remember that the edges also have some given lengths.

5.2. What is the running time of this algorithm, and why? (2 points)

Solutions (attached after the exam)

1.1. An initial set of such paths can be found in polynomial time, e.g., by BFS. We can get paths without self-crossings, that is, every path visits every node at most once. Let L be the sum of lengths of all chosen paths. Since the paths are not self-crossing, L is bounded by some polynomial of the graph size. In every step we get rid of two occurrences of some edge, and the remaining edges existed already in the solution before that step. It follows that L decreases by 2. This can happen only polynomially often, and every step needs only polynomial time, for finding a shared edge and for changing the paths. The algorithm stops only when no two paths share an edge anymore, that is, the final solution is always valid. (8 points)

1.2. Run the algorithm from 1.1, but in the end, check every path whether it is a shortest one. If not, replace it with a shortest path (which is computable in polynomial time). Clearly, every such step decreases L by at least 1. But the new patha may share edges again. Therefore we run the algorithm from 1.1 once more. This is iterated until all our k paths are shortest paths. Since L strictly decreases each time, the overall time remains poiynomial, as in 1.1. (4 points)

2.1. For $k = 1, \ldots n$ we proceed as follows. We start with $OPT(k) := 0$, accounting for a subsequence that starts with x_k and, thus, has no jumps so far. Every subsequence that has started earlier must have some last element x_i, $i < k$, before x_k. If $x_k - x_i \geq J$, we get a subsequence ending with x_k, and with $OPT(i) + 1$ jumps. Finally, $OPT(k)$ is the maximum of all these jump numbers. Since we consider all possible $i < k$, and it is pointless to extend a monotone subsequence without making a jump, we cannot miss any potential solution. (6 points)

2.2. Since n values $OPT(k)$ are computed, and for each of them, the maximum of $k-1 < n$ earlier values is taken, the time bound is $O(n^2)$. – Remark: One can combine dynamic programming with binary search and achieve an $O(n \log n)$ time bound, but this better result is not expected here. (3 points)

3. Let d denote the minimum length of the edges that do not cross the separating line S. We partition the stripe of width $2d$, with S in the middle, into squares of side length d, and find the shortest edge with end points in (directly or diagonally) neighbored squares at different sides of S. (No edges shorter than d can have their end points in remote squares.) Finally

we take the minimum of d and all these edge lengths. The overall time remains $O(n \log n)$ because the recurrence $T(n) = 2T(n/2) + O(n)$ applies. (10 points)

4. A fresh node cannot be in a triangle, since its two neighbors are not adjacent. Hence any triangle in H consists of nodes from G, but they are not adjacent either.
The time is polynomial (even linear), since every edge is processed only once.
Assume that G has an independent set I of k nodes. We construct a node set J as follows. Initially $J := I$. For every edge uv in G, let $u - x - y - v$ be the constructed path. At most one of u, v is in I, say $v \notin I$. We add y to J. Finally J has $k + m$ nodes and is independent.
Conversely, assume that H has an independent set J of $k+m$ nodes. Assume that two nodes $u, v \in J \cap V$ form an edge uv in G. Since J is independent, this means $x, y \notin J$, and we can replace v with y in J. Iterating this step we get rid of adjacent nodes in $J \cap V$ and make this set independent in G, while keeping J independent, too. At most m fresh nodes are in J, thus $|J \cap V| \geq k$.
The Independent Set problem is known to be NP-complete, and the construction fulfills all criteria of a polynomial-time reduction, hence the problem remains NP-complete even when restricted to triangle-free graphs.
(15 points)

5.1. We follow the suggested steps.
(a) For every i, a subtour starts and ends in r and visits all nodes of T_i. Hence the concatenation of these d tours is a tour that visits all nodes of T.
(b) The edge rr_i is obviously traversed exactly twice. Since this holds on every recursion level, this statement is also true for every edge in T.
(c) Let uv by any edge of T, where u is the last node on the unique path from r to v. In order to visit v, we must sometimes go from u to v. In order to return to r, we must sometimes go from v to u. Hence any solution has to use every edge at least twice.
(d) Due to (c), the minimum length of any valid tour is at least twice the sum of lengths of all edges. Due to (b), the proposed tour is no longer than that. (12 points)

5.2. The standard argument shows that the time is $O(n)$: Every edge is processed $O(1)$ times, and a tree with n nodes has $n - 1$ edges. (2 points)